Multiresolution volume processing and visualization on graphics hardware

van der Laan, Wladimir

First published as: Wladimir J. van der Laan, Andrei C. Jalba, and Jos B. T. M. Roerdink. Multiresolution MIP Rendering of Large Volumetric Data Accelerated on Graphics Hardware. In *Proc. Eurographics/IEEE VGTC Symposium on Visualization (EuroVis)*, pages 243-250, 2007.

Chapter 2

Multiresolution MIP rendering on graphics hardware

2.1 Introduction

This chapter is concerned with the development of efficient algorithms for Maximum Intensity Projection (MIP) on graphics hardware. MIP is a method frequently used to visualize volumetric data originating from magnetic resonance angiography (MRA) and ultrasound data. As scanner precision increases larger datasets are generated, for which a large amount of memory and processing capacity is consumed for representing and rendering these volumes. Hence, research is necessary into more efficient MIP rendering algorithms that maintain image quality, e.g., have a fixed error bound. An established method to obtain this goal is the use of multiresolution methods. Since the MIP transform is nonlinear, the standard linear multiresolution models based on wavelets [44, 73, 148] are not applicable. Instead, morphological pyramids involving nonlinear filtering operations can be used for multiresolution rendering.

We present novel MIP algorithms which exploit the programmability of modern graphics hardware. The class of algorithms which we investigate makes use of morphological pyramids as the underlying representation of the volumetric data. Our algorithms are based on the so-called Multiresolution Maximum Intensity Projection (MMIP) method developed by one of us [108, 109]. This method enables the user to visualize large datasets in real time with progressive refinement. After computing the pyramid representation in a preprocessing step, the pyramid levels can be projected individually for progressive rendering. In preview mode, the lower levels of the pyramid are projected first to show a coarse approximation, which can be quickly refined to the original on demand.

A disadvantage of projecting one level at a time is that the approximation quality improves in discrete jumps, as determined by the levels of the pyramid. To overcome this, *streaming MIP* was introduced in [109], which is based on resorting the detail coefficients of all pyramid levels simultaneously with respect to decreasing magnitude of a suitable error measure. All resorted coefficients are projected successively, until a desired accuracy of the resulting MIP image is obtained, thus allowing for continuous error control.

The main contributions are:

- An improved method for computing streaming MIP rendering

- A GPU implementation of the level-by-level MMIP and the streaming MIP algorithms, with the advantage that we can spread the load and the dataset over multiple graphics cards in a straightforward way, thereby achieving support for large volume data with an almost optimal speedup.

We perform comparisons with existing CPU and GPU techniques for MIP volume rendering on large datasets and show the improvement which we obtain by our approach.

2.2 Previous and related work

In the case of MIP a multiresolution decomposition scheme cannot rely on linear operations. Therefore, in [106] a morphological pyramid scheme was proposed for MIP volume rendering with progressive refinement. Such pyramids, which involve nonlinear spatial filtering by morphological operators, systematically split the volume data into approximation and detail signals [39]. As the level of the pyramid is increased, spatial features of increasing size are extracted. Morphological pyramids combine feature extraction with accelerated rendering in preview mode. A disadvantage of the above method is that the approximation accuracy makes a jump each time as an additional level of detail signals is taken into account. To allow for continuous error control, the streaming MIP method was proposed in [109] and it was found to outperform the MMIP method, both with respect to image quality with a fixed amount of detail data, and in terms of flexibility of controlling approximation error or computation time.

The type of morphological pyramid considered here is appropriate in the context of MIP because the morphological operation of dilation (involving the computation of maxima of voxel values in a local neighborhood) is compatible with the maximum computation involved in MIP, just as linear pyramids or wavelet representations [73, 148] are the right tool for the case of linear X-ray rendering. Even though the morphological operators are nonlinear and non-invertible, the pyramid scheme does allow perfect reconstruction. Therefore, after the pyramid has been constructed the original volume data can be discarded. Also, only integer computations are required.

To allow for compression domain rendering, it is essential to use a (fast) MIP implementation which can work directly on the data structures used to represent the pyramid. This can be achieved by a voxel projection method with an efficient volume data storage scheme [88], see section 2.4.1. This method is analogous to point-based rendering (PBR) which has been used for both surface and volume representations [114]. A major challenge of PBR algorithms is accurate interpolation between discrete point samples. We achieve this by means of an additional morphological closing of the output image.

Previous work on mapping elementary morphological operations to graphics hardware in the context of volume rendering and analysis can be found in [52].

2.3 Overview of the multiresolution MIP algorithm

We first define some elementary morphological operators [116]. Next we introduce morphological pyramids, in particular adjunction pyramids. Finally, we will discuss how morphological pyramids are used for efficient MIP rendering.

2.3.1 Morphological operators

Let f be a signal with domain $F \subseteq \mathbb{Z}^d$, and A a subset of $\mathbb{Z}^d$ called the *structuring element*. The *dilation* $d_A(f)$ and *erosion* $\varepsilon_A(f)$ of f by A are defined by

$$d_A(f)(x) = \max_{y \in A, x-y \in F} f(x-y), \tag{2.1}$$

$$\varepsilon_A(f)(x) = \min_{y \in A, x+y \in F} f(x+y). \tag{2.2}$$

Dilation and erosion simply replace each signal value by the maximum or minimum in a neighborhood defined by the structuring element A. The *opening* $\alpha_A(f)$ and *closing* $\phi_A(f)$ of f by A are defined by $\alpha_A(f)(x) = d_A(\varepsilon_A(f))(x)$, and $\phi_A(f)(x) = \varepsilon_A(d_A(f))(x)$. So openings and closings are products of a dilation and an erosion. The opening eliminates signal peaks, the closing valleys.

2.3.2 Pyramids

The general structure of (non)linear pyramids is as follows. From an initial (2-D or 3-D) data set f_0, approximations $\{f_j\}$ of increasingly reduced size are computed by a reduction operation:

$$f_j = \text{REDUCE}\,(f_{j-1}), \qquad j = 1, 2, \ldots L.$$

Here j is called the level of the decomposition. An approximation signal associated to f_{j+1} may be defined by taking the difference between f_j and an expanded version of f_{j+1}:

$$d_j = f_j \mathbin{\dot{-}} \text{EXPAND}\,(f_{j+1}). \tag{2.3}$$

The set $d_0, d_1, \ldots, d_{L-1}, f_L$ is referred to as a *detail pyramid*. Here $\dot{-}$ is a generalized subtraction operator (see below). Assuming there exists an associated generalized addition operator $\dot{+}$ such that, for all j,

$$\hat{f}_j \mathbin{\dot{+}} (f_j \mathbin{\dot{-}} \hat{f}_j) = f_j, \;\; \text{where } \hat{f}_j = \text{EXPAND}(\,\text{REDUCE}\,(f_j)),$$

we have *perfect reconstruction*, that is, f_0 can be exactly reconstructed by the recursion

$$f_j = \text{EXPAND}\,(f_{j+1}) \mathbin{\dot{+}} d_j, \quad j = L-1, \ldots, 0. \tag{2.4}$$

To guarantee that information lost during analysis can be recovered in the synthesis phase in a non-redundant way, one needs the so-called *pyramid condition*:

$$\text{REDUCE}\,(\text{EXPAND}\,(f)) = f, \text{ for all } f. \tag{2.5}$$

In the case of morphological pyramids, the REDUCE and EXPAND operations involve morphological filtering [39]. For the simplest case of the so-called adjunction pyramids [106], the morphological operators are the dilation $d_A(f)$ and erosion $\varepsilon_A(f)$ with structuring element A defined in (2.1) and (2.2), respectively. Then the REDUCE and EXPAND operators are denoted by $\psi_A^{\uparrow}$ and $\psi_A^{\downarrow}$, respectively, and have the form

$$\text{REDUCE} : \psi_A^{\uparrow}(f) = \text{DOWNSAMPLE}\,(\varepsilon_A(f)), \tag{2.6}$$

$$\text{EXPAND} : \psi_A^{\downarrow}(f) = d_A\,(\text{UPSAMPLE}\,(f)), \tag{2.7}$$

where the arrows indicate transformations to higher (coarser) or lower (finer) levels of the pyramid. Here DOWNSAMPLE and UPSAMPLE denote downsampling and upsampling by a factor of 2 in each spatial dimension. The pyramid condition (2.5) is satisfied, if there exists an $a \in A$ such that the translates of a over an even number of grid steps are never contained in the structuring element A; see [39] for more details.

In an adjunction pyramid, the product $\psi_A^{\downarrow}\psi_A^{\uparrow}$ is an *opening*. The anti-extensivity property of openings [49] implies that $\psi_A^{\downarrow}\psi_A^{\uparrow}(f) \leq f$. Therefore, we can define the generalized addition and subtraction operators $\dot{+}$ and $\dot{-}$ appearing in (2.3) by (cf. [39]):

$$t \mathbin{\dot{+}} s = t \vee s = \max(t, s), \qquad t \mathbin{\dot{-}} s = \begin{cases} t, & \text{if } t > s \\ 0, & \text{if } t = s \end{cases} \tag{2.8}$$

where 0 is the smallest image or voxel value possible. As a consequence, the detail signals are nonnegative:

$$d_j(n) = f_j(n) \mathbin{\dot{-}} \psi_A^{\downarrow}\,\psi_A^{\uparrow}\,(f_j)(n) \geq 0. \tag{2.9}$$

Note that the definition of $\dot{-}$ in (2.8) implies that the detail signal $d_j(n)$ equals $f_j(n)$, except at points n for which $f_j(n) = \psi_A^{\downarrow}\,\psi_A^{\uparrow}\,(f_j)(n)$, where $d_j(n) = 0$. So, detail signals are not 'small' in regions where the structuring element does not fit well to the data.

For an adjunction pyramid with the generalized addition being defined as the maximum operation (see (2.8)), the reconstruction takes a special form [108]:

$$f = \psi_A^{\downarrow L}(f_L) \vee \bigvee_{k=0}^{L-1} \psi_A^{\downarrow k}(d_k). \tag{2.10}$$

Here L is the decomposition depth, $\psi_A^{\downarrow k}$ denotes k-fold composition of $\psi_A^{\downarrow}$ with itself, and $\vee$ denotes the maximum operator.

2.3.3 Multiresolution MIP algorithm

The adjunction pyramid representation does allow to interchange the MIP operator (computing maxima along the line of sight) with the pyramidal synthesis operator, because both the upsampling operation and the dilation d_A commute with the maximum operation [106,108]. As a result, the MIP operation can be performed on a coarse level (reduced data size) before performing a

cheap 2-D EXPAND operation to a finer resolution, thus leading to a computationally efficient algorithm.

We write the MIP operation as $\mathcal{M}_{\mathbf{\Theta}}$, with $\mathbf{\Theta} = (\theta, \phi, \alpha)$, where θ and ϕ are the two angles defining the projection direction vector perpendicular to the view plane, and α gives the orientation of the view plane with respect to this vector. Successive approximations of the MIP of f are denoted by $\hat{\mathbf{M}}_{\mathbf{\Theta}}^{(j)}(f)$, $j = L, L-1, \ldots, 0$. These approximations all have the size of the MIP of the full data f in the image plane.

The MMIP algorithm for an adjunction pyramid is as follows. From a level-j approximation, the next approximation on level $j-1$ is obtained by first computing the MIP of d_{j-1}, then $j-1$ times applying the 2-D pyramid synthesis operator $\psi_{\tilde{A}}^{\downarrow}$ to the projection, and finally taking the maximum of the image so obtained with the previous approximation. Here $\psi_{\tilde{A}}^{\downarrow}$ is a 2-D EXPAND operator which has the same form as (2.7), that is, 2-D upsampling followed by a 2-D dilation, but with a structuring element $\tilde{A}$ which is the MIP of A, that is, $\tilde{A} := \mathcal{M}_{\mathbf{\Theta}}(A)$. It is clear that $\hat{\mathbf{M}}_{\mathbf{\Theta}}^{(j-1)}(f) \geq \hat{\mathbf{M}}_{\mathbf{\Theta}}^{(j)}(f)$, since from (2.9) the details signals d_{j-1} are nonnegative. So the projections increase pointwise as one goes down the pyramid.

2.3.4 Streaming MIP

Here we summarize the construction of the coefficient stream and the rendering for streaming MIP.

Construction of the detail coefficient stream. By the commutativity of the pyramidal synthesis operator with the maximum operation we can project the elements of the detail coefficients in any order on the image plane, not necessarily level by level, as we have done so far. So one can join the detail coefficients of all levels and sort these according to some error measure. We do not sort the detail coefficients $\{d_j\}$ directly. As can be seen from (2.9), the detail coefficients are not small in regions where the structuring element does fit approximately, but not exactly, to the data. Therefore an auxiliary set of detail coefficients can be defined which can be sorted with respect to decreasing magnitude. Then the original detail coefficients $\{d_j\}$ are resorted by giving them the same order as the resorted auxiliary coefficients, which are subsequently discarded; see [109] for details.

As a result of the construction phase, we have obtained an ordered list of detail coefficients $d_j(x, y, z)$, which can be used to compute the MIP of the input data. By construction, the order is such that each successive coefficient, when taken into account, maximally reduces the L_1-error between the partial reconstruction and the original data.

Rendering phase. In the MIP rendering phase, all sorted coefficients $\{d_j(x, y, z)\}$ are projected successively, until an a-priori chosen maximum number of coefficients has been projected, or a desired accuracy of the resulting MIP image is obtained. When a coefficient k is projected, its value val is compared to the current value $curval$ at the point of projection in the image plane, and $curval$ is overwritten when $curval < val$. Also, when the level of the coefficient is j, a local dilation of size j has to be carried out, i.e., all pixels in the scaled neighborhood $j \cdot \tilde{A}$ around the

point of projection are overwritten by val when their current value is smaller than val. Note that we cannot do the dilation globally, in contrast to the case of the level-by-level projection where the scale index is constant per level.

2.4 Implementation on graphics hardware

As discussed in section 2.3.2, the morphological pyramid of a dataset is built in a preprocessing step. For the MMIP projection which works level by level, each level is rendered separately using a MIP volume rendering method. Intermediate levels are rendered to a texture, starting from the coarsest level, after which the 2-D synthesis operator $\psi_{\tilde{A}}^{\downarrow}$ is applied and the result combined with the previous approximation, successively taking the detail signals into account. This two-dimensional synthesis operator is implemented as a fragment program which maps a $N \times M$ texture to a $2N \times 2M$ texture. The upsampling and dilation steps are rolled into one pass for efficiency. The levels are combined using the frame-buffer maximum operation. In what follows we discuss the separate steps in detail.

2.4.1 Per-voxel projection

To avoid processing empty space we use an object-order voxel-projection method [88], where one loops through the volume and projects all non-zero voxels in value-sorted order with each voxel contributing to exactly one pixel. By projecting the voxels from low to high value, old values in the image plane can simply be overwritten by current values. This allows for MIP rendering without expensive read-compare-write GPU cycles. This method also uses an efficient scheme for storing the volume data, based on histogram-based sorting of non-zero voxels according to their grey value. After the sorting step the voxels are represented by an array of positions. An additional array contains the cumulative histogram values. All levels of the pyramid are created and stored as value-sorted arrays.

2.4.2 Representing the detail coefficients

The voxel data are stored in a buffer that is created with the `ARB_vertex_buffer_object` extension. This extension defines an interface that allows data to be cached in high-performance graphics memory tailored to the use of these buffers, thereby increasing the rate of data transfers. A static buffer is requested that will be filled once by the application, and used many times as the source for GL drawing commands. The voxels of the volume are stored in a continuous region sorted per intensity value. The second structure, the histogram, is kept with the begin and end offsets in this buffer, for each intensity value. Voxels with one intensity level can then simply be sent to the vertex processor by invoking `glDrawArrays` once, with the begin and end values as found in this histogram.

The most naive implementation stores the intensity, x, y and z coordinate in shorts, and uses 8 bytes per voxel. The different attributes are provided to the vertex program as texture coordinates. Rows of consecutive voxels will have the same intensity value, as represented in the

histogram. Storing this with each voxel is very redundant, as the histogram acts as a kind of run-length-encoding. A shader constant or texture coordinate can be set to the intensity for each span of voxels with equal intensity. Now we are left with 6 bytes per voxel (2 bytes per coordinate). Theoretically this will allow for volumes up to 65536^3. In practice we cannot support such large volumes as they will not fit into memory on current hardware.

Let us assume that we want to reduce the memory requirements to 4 bytes per voxel (two `GL_SHORT`s). This means we have 32 bits available. Distributing this over X, Y and Z like $12+10+10$ ($4096 \times 1024 \times 1024$) will suffice for even huge volumes. We can unpack these in a vertex program. For GPUs that do not have bit shift and logical operators, these can be emulated with multiplication by a fraction (computed using the *floor* Cg function) and subtraction. Overall, this results in a major speedup for large volumes. Even though this unpacking requires some extra computation this is easily out-weighted by the savings in memory usage and the associated gain in the ratio of memory bandwidth to voxel count.

2.4.3 Projecting the detail coefficients

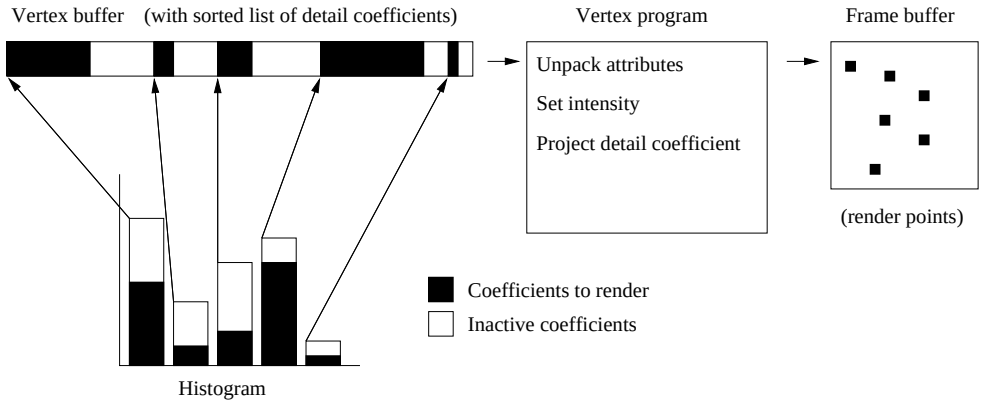

Figure 2.1. *The process of selection, projection and rendering of detail coefficients for one level of the pyramid. The white histogram is the histogram of the dataset, the (overlaid) black histogram shows the subset of the detail coefficients that is selected according to some the criterion.*

For the sake of clarity, but without sacrificing generality, we have used orthogonal projection throughout all examples. We build a model-view matrix from a quaternion which represents the orientation of the volume and a zoom factor (which defaults to 1 voxel on 1 pixel), both of which can be interactively adjusted. The projection matrix and the model-view matrix are then combined into a 4×4 model-view-projection matrix and passed to the vertex shader.

The entire volume is projected by iterating over the intensity values present in the volume, starting from the lowest (or from a user defined threshold below which everything is background) and stopping at the highest. If there is an upper intensity threshold above which everything has the maximum intensity, all these voxels can be projected with one call to `glDrawArrays`,

which is set to draw point primitives (without rounding or anti-aliasing). This is illustrated in Figure 2.1.

A multi-level representation is incorporated into the algorithm by storing the voxels for the different levels separately. One needs to store $L+1$ volumes for an L-level decomposition: detail levels $d_0 \dots d_{L-1}$ and the approximation f_L. These volumes are rendered in the same fashion as described already. Level 0 is rendered to a quad of the same size as the viewport, level 1 is rendered to a half-sized quad, level 2 to a quarter-sized one, and so on.

2.4.4 Load balancing

The MMIP algorithm has the advantage that we can spread the load and the dataset over multiple graphics cards in a straightforward way. In this way large volumes can be supported, and an almost optimal speedup is obtained. Fortunately, the newer NVidia drivers support multiple cards for one X-server for multiple screens. To render to both simultaneously, a `GL Context` is created on both X-screens, and rendering is switched between them one time per frame. Finally, the results of both cards are combined. Because the result of the second card is combined with that of the first, the second card does not necessarily have to be connected to a monitor at all. At the moment there is no extension for directly shuffling data between two cards, so the intermediate result will have to pass through the CPU. We found that using `glReadPixels` on one context and then `glDrawPixels` on the other one was the only currently available way of doing this. Our timings show that the preferred pixel format for this is `GL_UNSIGNED_INT_8_8_8_8`.

To utilize multiple cores or multiple processors, it was also attempted to put the rendering loop for each context in a different thread. This did not result in any performance gain. This is most probably because the algorithm is GPU bound, and the only processing done by the CPU is queuing commands and data to the GPU. Spreading this tiny load over multiple CPUs also does not outweigh the synchronization overhead.

The best performance was achieved by splitting the sorted array of voxels equally over both cards by interleaving the detail coefficients (also splitting the histogram) for each pyramid level. The coefficients are interleaved instead of being split in the middle to better divide the load in case of progressive refinement. In the end, the resulting images are combined using a pixel-wise maximum operator.

2.4.5 Streaming MIP

Streaming MIP can be implemented similarly using point rendering. For each voxel, we will have to store an extra attribute, namely the originating pyramid level. The level can be rolled into two bits of the position attribute (see section 2.4.2), which spans 4 bytes, and for intensity another short is required, making a total of 6 bytes per voxel.

To implement the level-dependent local dilation (see section 2.3.4) the size of the output voxel can be set using the point size extension to the ARB vertex program assembly language (which maps to the `PSIZE` output semantic in Cg). For example, assuming a square structuring element of size 2×2, the point size will be 1, 2 and 4 at level 0, 1 and 2, respectively. The specific shape of this point for each level depends on the structuring element A and the projection angle.

If the projection of A is not square, the *point sprite* extension is used to apply a specific shape to the point.

In addition to the increased memory usage, the fact that voxels are no longer ordered by intensity means that the writes to the frame buffer now have to be done with the maximum operator enabled. This results in some loss of performance, but not a huge one as the blended primitives are small. However, we will show next that this can be done in an even more efficient way.

2.4.6 Optimized streaming MIP

Note that the order in which the detail coefficients are rendered is not important, even though this method sorts them in a specific order. Therefore, the error-sorted list of details is not used to set the rendering order but only to guide setting the error for speed/quality tradeoff. The sorted array of detail coefficients, resulting from the preprocessing as required for streaming MIP, is subjected to a second histogram-sorting step with one bin per (level, intensity) pair. The ordering by error is maintained within these bins, having the bins store the position attribute of each detail coefficient. This results in a more efficient method for streaming MIP that avoids using point sprites and frame buffer blending, and which uses only four bytes per voxel.

The level and intensity attributes of the detail coefficients are kept in their original sorted order in one big list. The resulting histogram and sorted list can be used for rendering the entire dataset (or a per-level approximation) and also for streaming MIP rendering. This is implemented as follows. A percentage of detail coefficients to be rendered is chosen, after which the corresponding part of the sorted list is traversed. For each pyramid level the voxels are then rendered in a low-to-high intensity order, guided by the complete histogram and the number of coefficients that need to be rendered. This is the same algorithm as in section 2.4.1, except that an additional histogram is used to select which coefficients to render. With this new algorithm we have all the advantages of streaming MIP without any of the drawbacks mentioned in section 2.4.5.

2.4.7 Post-processing

For non axis-aligned parallel projections the voxel-based method can yield pixel-sized holes in the result. A post-processing step based on a morphological closing step with the structuring element $\tilde{A}$ used for synthesis can fill these holes effectively and efficiently.

2.5 Results

In this section we report some experimental results. All performance measurements were carried out on a machine with dual AMD Opteron 280 processor and two GeForce 7900GTX graphics cards. Unless mentioned otherwise, only one of the cards (and one of the CPUs) is active.

Table 2.1 shows the percentage of detail coefficients rendered for the Visible Woman dataset (dimensions $512 \times 512 \times 1734$, and a total of 210 million detail coefficients) versus three different

Table 2.1. *Approximation error as a function of the percentage of detail coefficients kept to render the VisibleWoman dataset (dimensions $512 \times 512 \times 1734$ and a total of 210 million detail coefficients). Three different error measures are shown (maximum, relative L_1 and median), and the performance is given in frames per second (FPS).*

Coeffs.	**Error**			**FPS**
(%)	max	relative L_1	$median$	
100	0	0	0	1.14
50	3	$9.6 \times 10^{-}5$	1	2.27
25	8	$2.0 \times 10^{-}3$	3	4.41
13	13	$1.2 \times 10^{-}2$	4	8.40
6	21	$2.6 \times 10^{-}2$	4	15.4
1	63	$1.0 \times 10^{-}1$	7	51.2

Table 2.2. *Approximation error as a function of the percentage of detail coefficients kept to render the XMasTree dataset (dimensions $512 \times 512 \times 512$ and a total of 105 million detail coefficients). Three different error measures are shown (maximum, relative L_1 and the median), and the performance is given in frames per second (FPS).*

Coeffs.	**Error**			**FPS**
(%)	max	relative L_1	$median$	
100	0	0	0	2.80
50	5	$1.3 \times 10^{-}4$	3	5.60
25	10	$5.8 \times 10^{-}4$	4	10.6
13	22	$1.8 \times 10^{-}3$	4	19.6
6	33	$7.2 \times 10^{-}3$	7	33.6
1	177	$7.0 \times 10^{-}2$	25	100.6

error measures, and the performance in frames per second. Table 2.2 shows the same results for the XMasTree dataset ($512 \times 512 \times 512$, and a total of 105 million detail coefficients).

In both tables, the *maximum* error measure (L_∞ norm) is the maximum difference in grey level between the original full quality MIP image and the approximation. However, this measure is not very representative for the perceived error. The *median* error measure is calculated by taking the median of the grey level differences (excluding the zero ones). The median error is expressed in grey levels and provides a good indication of how much the approximation differs from the original, not being very sensitive to outliers and noise. The relative L_1-approximation

Table 2.3. *Performance in frames per second (FPS) for the Visible Woman dataset in a* 512×512 *viewport for various MIP methods: (i) the most common texture-based volume rendering method; (ii) GPU ray-casting; (iii) MMIP, rendering everything but the highest detail level (iiii) streaming MIP, for an optimized software implementation, the GPU implementation (for two error settings), and for the workload distributed over two GPUs.*

Method	**Error** (L_1)	**FPS**
3-D Texture-based	-	1.0
GPU ray-casting [68]	-	2.0
MMIP, 2 levels (GPU)	0.16 (median 8)	8.0
Streaming MIP (software)	0.0	0.2
Streaming MIP (software)	0.07 (median 4)	3.5
Streaming MIP (GPU)	0.0	1.1
Streaming MIP (GPU)	0.07 (median 4)	18.4
Streaming MIP (2x GPU)	0.0	2.0
Streaming MIP (2x GPU)	0.07 (median 4)	30.2

error measure between the original image $\mathbf{M}^{(0)}$ and approximation $\mathbf{M}^{(j)}$ is defined as:

$$\varepsilon^{(j)} = \left\|\mathbf{M}^{(0)} - \mathbf{M}^{(j)}\right\| / \left\|\mathbf{M}^{(0)}\right\|,$$

where $\|\cdot\|$ represents the L_1 norm.

Visually, the error remains quite unnoticeable until the median error reaches a value around 5 to 7 (on the datasets which we experimented with), after which it generally starts to rise and artifacts appear. For interactive purposes a large reduction percentage (1-5 %) of the detail coefficients is acceptable, with a corresponding increase in performance. Also, it can be seen that the frame rate approximately doubles each time the number of detail coefficients is halved, so the relation between rendering time and amount of retained detail coefficients is linear.

Table 2.3 shows a comparison of various methods on the VisibleWoman dataset: (i) a brute force 3D texture-based method using view-aligned slices and frame buffer blending; (ii) GPU ray-casting [68]; and (iii) our streaming MIP implementations, both in software and on one or two GPUs, for various error levels. The render viewport was set to 512×512 in all cases. For both texture-based methods the dataset was split into four, three blocks of $512 \times 512 \times 512$ and one of $512 \times 512 \times 198$, because the hardware does not support 3D textures larger than 512^3.

When full reconstruction is performed, the streaming MIP in software achieves approximately the same performance as the 3D texture-based method. GPU ray-casting outperforms that method by making smart use of fragment programs and early ray termination. Streaming MIP starts to become interesting when allowing for a certain error. For example, when we admit a hardly visible error (L_1 error bound of 0.07, or median of 4 grey levels) we can achieve an interactive frame rate of 18 frames per second. Compared to the optimized software implementation, the GPU version is about six times faster given the same dataset and error bound. Using

two cards we achieve a speed up of almost a factor of two. In preview mode, a larger error may be acceptable (say median 7), which increases the performance to 50 FPS.

From this it can be concluded that our method works especially well on large datasets that do not fit into the memory of the graphics card(s) at once, and take too long to render using standard GPU ray-casting at the original resolution. With our method, such volumes can be rendered in real time, albeit a tradeoff between rendering quality and performance/resource usage has to be made.

Some MIP images obtained by our optimized method are shown in Figures 2.2 and 2.3. The first figure displays the MIP rendering of the XMasTree data set, for two different error levels. In the lower quality rendering (the left image) some degradation is visible in the background and at the base of the tree, but there is only a small difference in the tree itself. Figure 2.3 shows a rendering of the entire visible woman dataset at four error levels. The images on the first row show hardly any visible differences. However, the zoomed-in excerpts reveal some differences for strongly reduced coefficient percentages: at 5% the fine details of the ribs are still visible, whereas at 1% the image is visibly somewhat degraded.

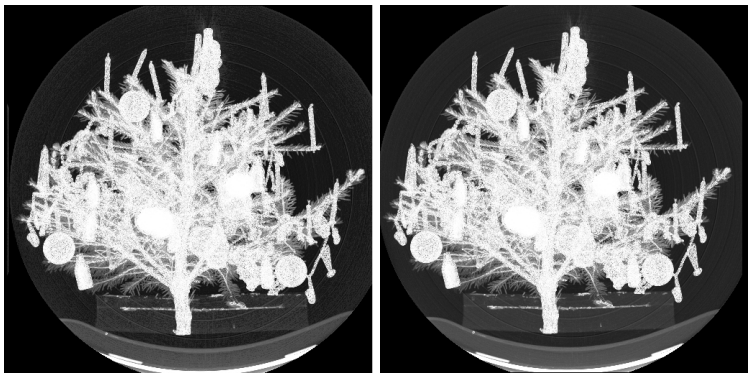

Figure 2.2. *MIP rendering of the XMasTree data set with 5% of the detail coefficients (left image) and at full reconstruction (right image).*

2.6 Discussion

A number of issues arise which require some further comments.

The sorting step during pyramid construction can take some time depending on the number of non-zero voxels, which is up to five minutes for the VisibleWoman dataset. Normally this will not be a problem, but if there is a hard time constraint in processing incoming data the method is unsuitable. A GPU sorting method can be used to accelerate this step. The most efficient method we are aware of was introduced by Gress and Zachmann [40], and is based on adaptive bitonic sorting (complexity $O(n \log n)$). For sorting n values utilizing p stream processor units, this approach has the optimal time complexity $O((n \log n)/p)$. On recent GPUs, this approach has shown to be remarkably faster than sequential sorting on the CPU.

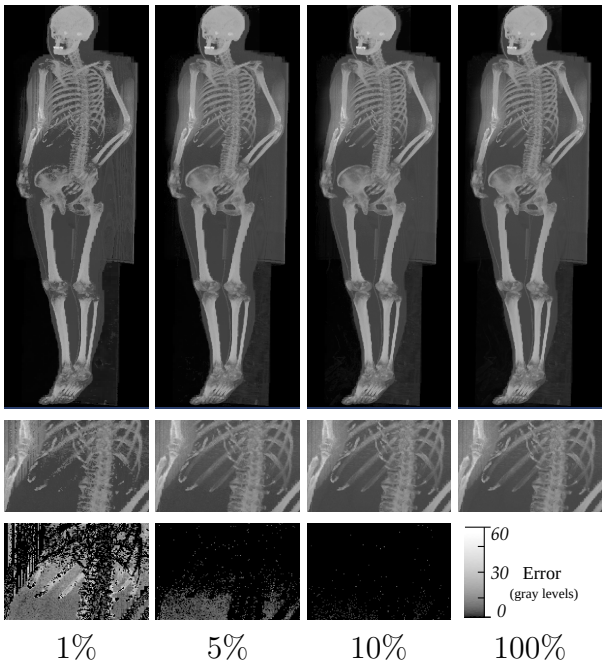

Figure 2.3. *MIP rendering of the complete Visible Woman dataset in a* 800 *by* 2000 *window (using streaming MIP Projection). The rendering is shown at various quality settings (given as percentages of the total number of detail coefficients). The second row shows a detail image for each quality setting, and the third row shows the difference image in gray levels.*

Current graphics hardware does not have enough memory to accommodate very large datasets. For this method, this is less of a problem as one can decide to only store the most important detail coefficients. With the upcoming generation of Shader version 4 GPUs, virtualisation of the memory is possible such that GPU memory is mapped on the host system. This will lessen the memory problem, but the increased access time of external memory can still make it worthwhile to use an streaming method like this.

Even though point rendering on the current generation of GPU hardware is faster than the equivalent on the CPU, the algorithm is bound by the vertex units of the GPU, which are still slower and less in number than the fragment units. This problem will disappear with the upcoming hardware which has unified shader units. This will give a large improvement to rendering speeds as the graphics hardware can allocate all of the shader units to process points.

Throughout this chapter a three-level pyramid (two detail levels and one approximation level) was used. It is also possible to use pyramids with more levels but we found no improvement in image quality or rendering speed with the data sets that were used. Performance even decreased a bit because more render, upsample, dilation steps are needed. The reason for no gain appears to be that structuring elements become larger than any structure present in the data, so that the higher levels have little non-zero coefficients. For even larger datasets the results might improve with more levels.

No attempt was made to remove voxels that do not contribute to the image from any viewing

angle. According to [88], a view-independent hidden voxel removal step is able to remove about half of the voxels in a dataset, at the cost of more preprocessing. Assuming this, a speedup in rendering by another factor of two could be achieved.

2.7 Conclusion

We have investigated a number of algorithms based on morphological pyramids for multiresolution MIP volume rendering on graphics hardware. We found that our highly-optimized streaming MIP GPU-method outperforms both its software implementation as well as existing ray-casting and 3-D texture-based methods.

Using the basic texture-based MIP method for each level, then synthesizing the result is the most obvious way of implementing multilevel volume rendering. But it is, by definition, not faster than plain volume rendering, as it does not take much advantage of storing only necessary voxels and near-continuous error control.

Per-voxel projection, as discussed in this chapter, is more advantageous. It applies point-based rendering to project the dataset a voxel at a time, and implicitly uses empty space skipping by only rendering non-empty voxels. The user can interactively adjust thresholds and set the performance/quality ratio as desired. The algorithm is also cache efficient, as it always addresses GPU memory in a linear way. In addition, the load and the dataset can be divided over multiple GPUs to achieve a near-optimal speedup, even for large volume data.